Real-time Safe Interval Path Planning

Devin Wild Thomas[1], **Wheeler Ruml**[1], **Solomon Eyal Shimony**[2]

[1]University of New Hampshire, USA
[2]Ben-Gurion University of the Negev, Israel
{dwt, ruml}@cs.unh.edu, shimony@cs.bgu.ac.il

Abstract

Navigation among dynamic obstacles is a fundamental task in robotics that has been modeled in various ways. In Safe Interval Path Planning, location is discretized to a grid, time is continuous, future trajectories of obstacles are assumed known, and planning takes place offline. In this work, we define the Real-time Safe Interval Path Planning problem setting, in which the agent plans online and must issue its next action within a strict time bound. Unlike in classical real-time heuristic search, the cost-to-go in Real-time Safe Interval Path Planning is a function of time rather than a scalar. We present several algorithms for this setting and prove that they learn admissible heuristics. Empirical evaluation shows that the new methods perform better than classical approaches under a variety of conditions.

Introduction

Safe Interval Path Planning (SIPP) (Phillips and Likhachev 2011) is a popular way to formalize the problem of navigation among moving obstacles. In SIPP, the agent and obstacles move on a discrete grid. Time is continuous and the obstacles cause cells to be blocked during certain intervals and 'safe' during the complementary intervals. The trajectories of obstacles are assumed to be known for as far into the future as necessary and the agent plans off-line, before any obstacles start moving.

In this paper, we drop the assumption that the world stands still while the agent plans. We call this new problem Real-time Safe Interval Path Planning (RTSIPP). The planner is given a pre-specified time bound and must return the next action for the agent to take within that time. The agent may be unable to find a complete path to a goal location within that time, so planning iterates, returning one action at a time, until the agent reaches a goal. In each iteration, the planner looks ahead as far as possible within the time bound, then moves towards the frontier node n that yields the shortest estimated plan duration. This duration, notated $f(n)$, is estimated as the sum of the time to reach n, notated $g(n)$, and the estimated time to reach a goal from n, $h(n)$.

Note that $h(n)$ is just an estimate and the returned action is not guaranteed to lead towards a goal. A node n with promising $f(n)$ may in fact lead to poor successors. After moving to a successor node y, it is possible that the agent may conclude that n is the most promising successor of y, creating an infinite cycle. For this reason, learning has long been recognized as a core part of real-time planning: the planner updates the h values of locations based on its lookahead, as every non-goal state is only as good as the best path it enables through the lookahead frontier. Learning improved heuristic values allows real-time planners to guarantee reaching a goal under certain conditions (Korf 1990).

However in RTSIPP problems, unlike in the standard real-time search setting, the time-to-goal is not a simple scalar: due to the moving obstacles, the best path will depend on when the agent departs. While this issue can be ignored in classic offline SIPP because execution always starts at time 0, it plays a crucial role in RTSIPP because the agent might later consider returning to a previously visited state and it could be prevented from reaching a goal if it erroneously generalized that a cell that was temporarily blocked by an obstacle were always inaccessible. To be correct, heuristic learning must be more complex.

Recent work on any-start-time planning showed how to represent g values in offline heuristic search as functions of time, so-called 'arrival time functions' (ATFs) (Thomas et al. 2023). In this paper, we show that this same representation can be used for g, h, and f values in real-time heuristic search. We present variants of RTA* (Korf 1990) and PLRTA* (Cannon, Rose, and Ruml 2014) that provably learn admissible heuristics and generalize correctly in RTSIPP. Results of an experimental evaluation show that our new methods outperform naive strategies for handling dynamic obstacles in real-time planning. In addition to providing a more realistic version of SIPP, this work also provides a foundation for handling dynamic environments more effectively in heuristic search.

Background

Our work builds on previous research in real-time heuristic search, SIPP, and any-start-time planning.

Real-time Heuristic Search

Real-time heuristic search was introduced by Korf (1990). We can understand most real-time search algorithms as iterating three stages: First, in the planning stage, the agent plans for a fixed time budget. In so-called 'agent-centered search', this takes the form of a partial search tree rooted at the agent's current state (Koenig 2001), known as the local search space (LSS). Second, in the learning stage, the agent updates h values by backing up values from the search frontier. Third, in the commitment stage, the agent uses the updated f values to choose actions to commit for execution.

Real-time A* (RTA*) (Korf 1990) plans using a fixed-depth lookahead, learns by updating the heuristic value of the current state from the search frontier, and commits to the top-level action (applicable at the current state) with minimum f. Most relevant for our purposes, RTA* only updates the heuristic value for a single state at each iteration. A parent p's heuristic value becomes the minimum over all its children $succ(p)$ of the cost to reach the child plus the child's heuristic value:

$$h(p) \leftarrow \min_{c \in succ(p)} cost(p, c) + h(c) \tag{1}$$

Local search space learning real-time A* (LSS-LRTA*) (Koenig and Sun 2009) plans using a node-limited A* search, resulting in a search frontier like RTA*, though it is unlikely to have a fixed depth. To learn, LSS-LRTA* uses a Dijkstra-like algorithm to update the heuristic values of all states in the LSS by propagating heuristic information from the search frontier. Vanilla LSS-LRTA* commits to the entire sequence of actions up to the frontier node with lowest f; a variant that commits only to the best *top level action*, which is the first action in that sequence, was subsequently found to perform better (Kiesel, Burns, and Ruml 2015).

In a static setting, in a finite problem space with positive edge costs, and with a goal reachable from every state, both RTA* and LSS-LRTA* are complete. However, this does not necessarily hold in a dynamic environment. In problems like SIPP, time must be part of the state, because the applicable actions at a location depend on the location of the dynamic obstacles, so the state space is technically infinite. For example, consider a food delivery robot waiting at a pedestrian crossing. If the light says 'do not cross' (thus the crossing is not traversable), we are confronted with the challenge of how to generalize the learning of that information. It is correct but almost useless for the robot to learn that at that location and exact moment the crossing is not traversable. A moment later, it might waste effort checking the crossing again. If it instead learns that the crossing is blocked and generalizes that to all time, then it has learned an inadmissible heuristic that could render the search incomplete.

Partitioned Learning Real-time A* (PLRTA*) (Cannon, Rose, and Ruml 2014) addresses this by partitioning the heuristic value of a state into two components: a static component h_s corresponding to the part of h due to the static environment, that is shared among all states that differ only in the time, and the dynamic component h_d, corresponding to any increased cost-to-go of this specific state that is caused by dynamic elements of the environment. PLRTA* updates h_s as in LSS-LRTA*, using the g_s of the nodes in the LSS, and likewise for h_d. PLRTA* considers moving obstacles whose location is represented as a bivariate Gaussian distribution, coming for example from a noisy observation of the obstacle by the robot agent. PLRTA* handles the inadmissibility issue by linearly decaying the dynamic component of the heuristic back to zero. Thus in the crossing example, the robot would observe the 'do not cross' light, and explore for a while until it forgot about the crossing being impassable because that learned dynamic heuristic had decayed back to 0, and potentially return to explore it again, which is undesirable. PLRTA* has no completeness or correctness guarantee when there are moving obstacles present, only inheriting the completeness guarantee of LSS-LRTA* in a purely static environment.

Suppose we knew the timing of the light. We would be able to generalize better than PLRTA*, as we could know exactly when the light would change, rather than relying on the agent slowly forgetting that the crossing was infeasible. To our knowledge, no existing learning method correctly generalizes for a real-time search in a SIPP state space.

Safe Interval Path Planning

Safe interval path planning (Phillips and Likhachev 2011) is both a problem setting and a state space representation for efficient planning with continuous time. The states and actions are either safe or unsafe at every point in continuous time. All future dynamic changes to the environment are known. In the SIPP state space, times at which a location or action is safe are grouped to form a safe interval. Safe intervals are maximal, in that the location or action is unsafe immediately before and immediately after the interval. (We use the term safe to maintain consistency with prior work, but this representation also applies where other terms such as valid, allowed, or applicable would be more accurate.)

Formally, a SIPP problem is a tuple $\langle S, E, \delta, s_o, x_g \rangle$ where the states $\langle x, i \rangle \in S$ have configuration x and time interval $i = \langle t_b, t_e \rangle$. Similarly edges $\langle u, v, i \rangle \in E$ have source state u, destination state v, and interval i when it is safe to depart from u to v. The safe intervals i represent a maximal set of consecutive times where it is safe or otherwise possible to be in the associated state or depart along the associated edge. It is unsafe for an agent to be in a configuration while not covered by a safe interval or departing along an edge outside of a safe interval. The cost of an edge is its duration $\delta(x(u), x(v))$, which is time-invariant. The start state s_o is the configuration and interval the agent will start at and the objective is to find a plan that arrives at the goal configuration x_g as early as possible. A solution is a sequence of safe actions for the agent providing a safe path from the start to the goal. The objective is to provide a solution that minimizes the arrival time of the agent at the goal.

Phillips and Likhachev (2011) propose a search algorithm, also named SIPP, that performs A* search over SIPP states, where $g(n)$ is the earliest known time at which the agent may arrive at the configuration represented by n within the corresponding safe interval. Paths arriving at n later are dominated and pruned. An admissible heuristic $h(n)$ returns a non-overestimation of the time-to-goal from n. A successful SIPP search finds a single optimal path, arriving as early as possible at each intermediate step along the path.

The crossing example becomes trivial when re-framed as a SIPP problem. There are two states, s_o: the side of the crosswalk the robot starts on at $t = 0$ and s_g: the far side, which are both always safe. The unit edge e that connects

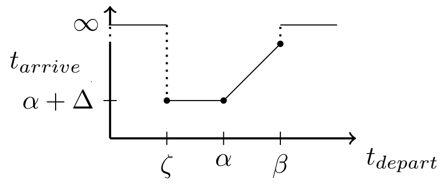


Figure 1: An edge ATF with parameters ζ, α, β, and Δ.

them has a safe interval $i(e) = \langle 10, \infty \rangle$ when the robot is allowed to traverse the crosswalk. A SIPP search will perform a single expansion of s_o, generating a successor at s_g with $g = 11$ corresponding to a plan that crosses the crosswalk the moment that the robot is allowed to. Suppose that our robot is planning online as part of a real-time search as it tries to reach the delivery location. The SIPP search has provided us with what we need to commit to our agent's next action. However, the scalar g we used to search is inadequate for supporting the learning required for real-time search. The scalar g does not track any information on how long the plan it represents remains safe. This limits our ability to generalize, without being overly conservative while learning a scalar heuristic.

Any-start-time Planning

Any-start-time planning (Thomas et al. 2023) plans for all start times, rather than a single start time. To do this, g is a function that returns the earliest arrival time for any safe departure time from the initial state. Such functions are called arrival time functions (ATF). Revisiting our crossing example, the any-start-time plan would have a constant arrival time of 11, until $t = 10$ after which the ATF is $t + 1$. Figure 1 shows the standard form of an ATF for a path p in a SIPP state space. It is a piecewise linear function $A[p]$ that is parameterized by four real-valued numbers: $\langle \zeta, \alpha, \beta, \delta \rangle$. The critical departure times are: ζ when the start state of the path begins, α is the earliest departure time where the agent would not be forced to wait along its path, and β the latest departure time before one of the states or edges becomes unsafe. Δ is the duration of actual movement along the path. These parameters define the piecewise linear function:

$$A[\zeta, \alpha, \beta, \Delta](t) = \begin{cases} \infty & t < \zeta \\ \alpha + \Delta & \zeta \le t < min(\alpha, \beta) \\ t + \Delta & \alpha \le t < \beta \\ \infty & \beta \le t \end{cases} \tag{2}$$

There is a constant arrival time segment from ζ to α with arrival time $\alpha + \Delta$, after which the ATF increases linearly in the departure time. The equation allows the counter-intuitive case where $\beta < \alpha$, which is necessary because it is not always possible to do all of the waiting required by a path in the initial state of the path. In such a case, the second case of Equation 2 applies, and the arrival time function for a safe departure time ($\zeta \le t < \beta$) is $\alpha + \Delta$.

Augmented SIPP (ASIPP) is an adaptation of SIPP to search with ATF g rather than a scalar g. The augmented SIPP search states consist of a SIPP state (configuration x and safe interval i) augmented with an ATF A storing the earliest arrival time from the start to that SIPP state. ASIPP produces an ATF of the found path, rather than a single scalar earliest arrival time. The ASIPP search works by compiling the safe intervals of the SIPP graph into an augmented SIPP graph, identical to the SIPP graph except with an ATF for each edge on the graph. From two SIPP states (u, v) and the edge x joining them, we can calculate the four parameters of the edge ATF:

$$\zeta = t_s(i(u)) \tag{3}$$

$$\alpha = max(t_b(i(x)), t_b(i(u)), t_b(i(v)) - \delta(u, v)) \tag{4}$$

$$\beta = min(t_e(i(x)), t_e(i(u)), t_e(i(v)) - \delta(u, v)) \tag{5}$$

$$\Delta = \delta(u, v) \tag{6}$$

When we search with an ATF g, the g of a child being generated is no longer the sum of a scalar parent g and scalar edge cost. Instead, ATF at the child is the composition of the ATF of the parent and the ATF of the edge. The structure of Equation 2 (shown in figure 1) is maintained under composition. If we have path p consisting of edge e followed by e', the ATF of p is:

$$A[p] = A[e'] \circ A[e] \tag{7}$$

which has the equivalent edge ATF, $A[p]$ with parameters:

$$\zeta_p = \zeta_e \tag{8}$$

$$\alpha_p = max(\alpha_e, \alpha_{e'} - \Delta_e) \tag{9}$$

$$\beta_p = min(\beta_e, \beta_{e'} - \Delta_e) \tag{10}$$

$$\Delta_p = \Delta_e + \Delta_{e'} \tag{11}$$

We denote functions that produce scalars using parenthesis, and functions that produce ATFs with brackets. For example, $g[n]$ in augmented SIPP is an ATF but $g(n)$ in regular SIPP is a scalar. We will generally drop the (t) from ATFs unless they are being evaluated at a specific time t to produce a scalar. To order open, we can use the ATF of the node instead of g to calculate a scalar $f(n) = \alpha(g[n]) + \Delta(g[n]) + h(n)$. These elements (ATF g, successor generation through composition, and node ordering on earliest arrival time) are the augmentations that turn SIPP into ASIPP. Thomas et al. (2023) find that there is minimal overhead associated with ASIPP compared to a regular SIPP search. To compute an any-start-time plan, Thomas et al. (2023) repeatedly runs ASIPP with a monotonically increasing starting time, storing the resulting ATFs in a set of non-dominated ATFs, called a compound ATF. The compound ATF can be efficiently queried for a plan at any-start-time. This is done by maintaining an ordered set of intervals, each pointing to the search node (with ATF g) that is dominant for all times in that interval. A node n is dominant at time t if $g[n](t) \le g[n'](t)$ for all other n' in the compound ATF.

Real-time SIPP: Definition and Challenges

Formally, an RTSIPP problem is a tuple $\langle S, E, \delta, s_o, x_g, b \rangle$, just as in offline SIPP except for the addition of the time budget b for each planning iteration. A solution to RTSIPP is a sequence of actions emitted over time, with action i emitted at time $i \times b$ after the start of planning. The objective is

Algorithm 1: Node-limited Augmented SIPP

```
1:  function NLASIPP(start, goal, budget)
2:      open ← $\{start\}$, cl ← {}                ▷ cl: ‘closed list’
3:      while open not empty do
4:          cur ← open.pop()                 ▷ $\min f = g(t_0) + h$
5:          if cur is goal or budget exhausted then
6:              return open $\cup\{cur\}$, cl
7:          for $e \in$ successors(cur) do
8:              $\alpha$ ← max($\alpha$(e) - $\Delta$(cur), $\alpha$(cur))
9:              $\beta$ ← min($\beta$(e) - $\Delta$(cur), $\beta$(cur))
10:             $\zeta \leftarrow \zeta(cur)$, $\Delta$ ← $\Delta$(cur) + $\delta$(e)
11:             g ← $A[\zeta, \alpha, \beta, \Delta]$, d ← destination(e)
12:             if d $\notin cl$ or $\alpha + \Delta$ <cl(d) then
13:                 cl[d] ← $\alpha + \Delta$
14:                 open ← open $\cup\langle$d, $g, h(d), cur\rangle$
15:     return Failure
```

to minimize total solution duration, subject to keeping the agent safe.

We begin by presenting Node-limited Augmented SIPP (NLASIPP), a node-limited version of ASIPP used by our new methods for RTSIPP. We then present baseline approaches to solving RTSIPP with LSS-LRTA* and partitioned learning, each backup a scalar heuristic. We discuss the shortcomings of the baseline approaches and present our three novel real-time heuristic search algorithms that use ATFs to solve RTSIPP which address these shortcomings.

Node-limited Augmented SIPP

Node Limited Augmented SIPP (NLASIPP, Algorithm 1) is a straightforward variant of ASIPP that reduces to ASIPP when budget = ∞. NLASIPP takes as input a start state, a goal configuration, and an expansion budget. Like augmented SIPP, NLASIPP is an A* search on the augmented SIPP graph, using a path ATF of the form of Equation 2 instead of a scalar-valued g, as shown in the successor generation (line 8-10). Search nodes are a tuple $\langle s, g(t), h, p\rangle$, where $s \in S$ is the SIPP state, $g(t)$ is the ATF, h is the scalar heuristic cost-to-go, and p is the parent pointer. The earliest arrival time corresponding to the scalar-valued g used by SIPP is $\alpha + \Delta$, which is summed with h to calculate a scalar f to order nodes on open (line 4). The earliest arrival at each state is stored in the closed list (line 13) and used to prune successors, as it is always at least as good to arrive earlier in a state (line 12). Because we use NLASIPP as a component of a real-time heuristic search, it returns the *open* and *closed* lists (line 6) for use by learning and commitment strategies.

We define the shifted Identity ATF, notated as:

$$I[shift](t) = t + x = A[-\infty, -\infty, \infty, shift](t) \qquad (12)$$

which for all finite t triggers the third case of Equation 2. $I[0](t)$ is the ATF for a path of no edges, and we define it for use as a base case. NLASIPP inherits an important property of ASIPP, which is that all optimal paths to a goal are represented by a node on *open* with a path ATF that is a non-overestimate of the cost of the path:

Theorem 1. *Given a SIPP problem $\langle S, E, \delta, s_o, x_g\rangle$ and any admissible scalar heuristic h over S, then always in line 3 of NLASIPP using h, for all $t \in i(s_o)$, and any optimal path $p*$ from s_o to x_g departing at time t, there exists a node n_f on open, with frontier state $S(n_f)$, such that: a) $S(n_f) \in p*$ and b) $I[h](A[n_f](t)) \leq A[p*](t)$.*

Proof. By induction on steps of the augmented SIPP search. When open contains only the initial node: s_0 must be on the path so (a) holds. The path ATF is the identity and the heuristic is admissible so (b) holds. The inductive step is successor generation, during which a child $s'_f \in p*$ is generated. s'_f can only be pruned if a state arriving earlier in s'_f has already been generated. If this state is on open, then the inductive step for (a) holds. Otherwise, s'_f is closed which would contradict $p*$ optimal unless (a) and (b) hold for some state later in $p*$. Cost is monotonic so (b) holds during successor generation. □

Scalar Learning for RTSIPP

We now describe two baseline approaches that are straightforward adaptations of LSS-LRTA* and PLRTA* to the SIPP state space. For each, we show a simple SIPP problem that demonstrates its shortcomings, which will be addressed by our ATF-based methods. The first baseline uses the learning method of LSS-LRTA* to backup a single scalar heuristic h. We will refer to this specialization as LSS-SIPP, to avoid confusion when referring to the general concepts of LSS-LRTA*. In order for the learned h to be admissible at all times t within the SIPP state, we need to also minimize over departure time in the LSS-LRTA* learning backup (Equation 1) for successor edge c with ATF A:

$$h(p) \leftarrow \min_{c \in succ(p)} \min_{t}(A[c](t)) + h(c) \qquad (13)$$

If we did not minimize over departure time, our heuristic could be inadmissible for some possible departure times.

There are two issues with minimizing over t to learn a scalar heuristic, rather than learning h as a function of t. The first issue is that if we learn a scalar heuristic value for each interval, that learning is not generalized beyond that interval. To address this, our second baseline, learns a partitioned heuristic, with one of the partitions being the time-independent heuristic h_s that is valid for all times. We call this second baseline PLRTS. Learning the partitioned static heuristic addresses the first issue. However, because every state with a successor by definition has a departure time that requires no waiting, the dynamic component of the scalar heuristic learned by PLRTS is always 0.

The second issue arises particularly for long intervals. Because we are required to minimize over all possible departure times, our learned scalar heuristic will tend to be overly optimistic. For example, consider the RTSIPP problem shown in Figure 2a that has an expansion budget $b = 1$ and three states: S the start state, G the goal state, and X representing a detour. All states are always safe. The edge, $\langle S, X\rangle$ takes one time unit and is always safe, and $\langle X, G\rangle$ takes ten time units and is also always safe. Finally, $\langle S, G\rangle$ takes one time unit but is unsafe until $t = 80$. An LSS-SIPP

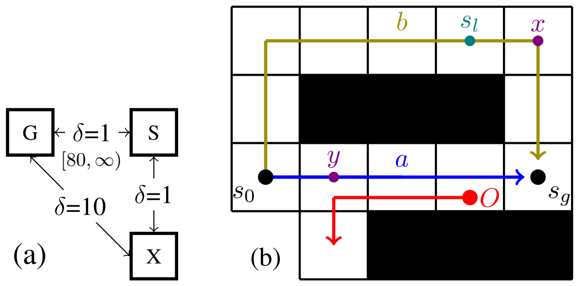


Figure 2: RTSIPP examples that defeat scalar learning.

agent will learn a heuristic for S of $h(S) = 1$ from the edge directly to G at $t \geq 80$. It will then commit to traveling to X. On the second iteration, it will learn $h(X) = 2$ and commit to S which has $f = 2$. On later rounds, it will continue to cycle between S and X with no additional heuristic learning. The agent's learning is unable to encode that at early times the detour is preferable and it takes until $\langle S, G\rangle$ becomes safe for the agent to escape.

Exact Learning for Real-time SIPP

We now present three approaches that, instead of relying on scalar approximations, learn exact representations of cost-to-go for RTSIPP. First, some preliminaries.

All of our ATF-based methods perform partitioned learning. PLRTA* has scalar $f(n) = g(n) + h_d(n) + h_s(n)$. In RTSIPP, $h_d(n)$ should clearly be a function of time, while $h_s(n)$ remains a scalar. However, with ATFs it is clearer to learn the function f, where:

$$f[n](t) = h[n](g[n](t)) = g[n](t) + h_d[n](t) + h_s(n) \quad (14)$$

This means that f is not initially 0, but instead $I[h_s](g(t))$. Note that $f[n]$, $h[n]$, and $h_d[n]$ are compound ATFs, $g[n]$ is an ATF, and h_s is a scalar. The ATF $g[n]$ corresponds to the duration of the path up to the state in node n, while $f[n]$ is the heuristic cost to the goal from the state in node n plus $g[n]$. Conceptually each ATF in $f[n]$ can be divided into two components, the ATF to a search frontier state from a prior round of search plus the static heuristic at that state. That sum is what we learn, which is why we refer to learning the summed heuristic f, though h or h_d can be retroactively calculated from f, g, and h_s if desired.

Minimal ATF Learning for RTSIPP

Our first approach, Real-time Augmented SIPP (RTAS) is an RTA*-like approach, in that it limits learning to only the current state. This allows RTAS to directly use Theorem 1 and transform the open list returned by NLASIPP into a compound arrival time function for the current state. This means that it learns directly from the open list of the search, without requiring any backing up through the intermediate nodes in the closed list.

Algorithm 2 shows the pseudocode of RTAS. NLSIPP (line 4) returns the search frontier when the budget is exhausted. In line 6 we initialize the scalar static heuristic to an overestimate, ∞. The dynamic heuristic is initialized to the empty set of ATFs if the current state has never been visited before (line 6). The static component of the learned heuristic is the minimum sum of the cost to the frontier and the heuristic time-to-goal at the frontier (line 9). Recall that we use parentheses for scalars such as $h_s(x)$ and square braces for ATFs such as $A[node]$ and $f[s]$ to disambiguate evaluating the ATF function. To compute the dynamic heuristic, each path ATF $A[node]$ is shifted later by the static heuristic of the frontier node (line 10), then inserted in the compound ATF $f[s]$ (line 11). Finally, because they are both admissible, we take the maximum of our old static heuristic and the new one (line 12) and commit to the best top-level action, returning it in line 13.

Algorithm 2: Real-time Augmented SIPP

```
1:  function RTAS(cur, goal, b)
2:      if cur is goal then
3:          return success
4:      open, closed = NLASIPP(cur, goal, b)
5:      s = state(cur), x = configuration(s)
6:      $h'_s(x) = \infty$ , $f[s] = \{\}$
7:      for node ∈ open do
8:          nx = configuration(state(node))
9:          $h'_s(x) = min(h'_s(x), \Delta(g[node]) + h_s(nx))$
10:         $\Delta(g[node]) = \Delta(g[node]) + h_s(nx)$
11:         $f[s] = f[s] \cup g[node]$
12:     $h_s(x) = max(h'_s(x), h_s(x))$
13:     return BESTTLA(open)
```

Figure 2b shows an example search from an RTSIPP problem. The agent is navigating from s_o to s_g, path a is blocked temporarily by moving obstacle O, states: x, y $\in$ $open$, s_l $\in$ LSS. Our agent in state s_o has done an NLASIPP search for a path to the goal at s_g. There are two possible paths, a is shorter in distance, but requires waiting on a slow-moving obstacle O, while b is longer but requires no waiting. The agent moves with unit speed, while the obstacle O takes six units to clear path a. For this example, let's say that NLASIPP returns open consisting of states x and y (and the closed list, which is not used by RTAS).

Figure 3 shows the dynamic heuristic learned by RTAS for s_o. The open nodes have the path arrival time functions $A[x]$ and $A[y]$, plotted in dotted violet. $A[x]$ has traveled

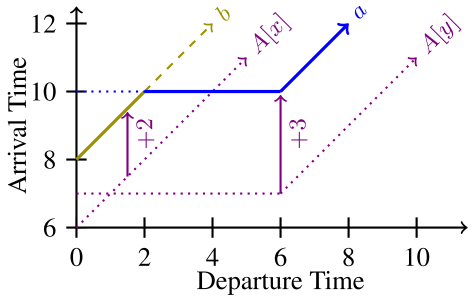


Figure 3: Compound ATF from example in Figure 2b.

Algorithm 3: LearnStatic

```
1: function LEARNSTATIC(open, closed)
2:     for state in closed do
3:         $h_s(state) \leftarrow \inf$
4:     order open by increasing $h_s$
5:     while open not empty and closed not empty do
6:         n ← pop(open)
7:         if state(n) in closed then
8:             remove state(n) from closed
9:         for all parent $p$ of n do
10:            if $h_s(p) > \delta(p,n) + h_s(n)$ then
11:                $h_s(p) \leftarrow \delta(p,n) + h_s(n)$
12:                Place p on open, maintaining sort
```

six units with no waiting, with $\zeta = 0$ its ATF has parameters $\langle 0, 0, \infty, 6\rangle$ and, for example, an arrival time of 6 if the agent departs at time 0. On the other hand, $A[y]$ must spend six units waiting for O to get out of the way, plus one unit traveling, giving parameters $\langle 0, 6, \infty, 1\rangle$ and an arrival time of 7 unless the agent departs on or after 6. Using Manhattan distance as the static heuristic, $h_s(x) = 2$, and $h_s(y) = 3$. The compound ATF $f[s_o]$ is the minimum of the open ATFs shifted by their static heuristic, with the best known, (and in this case, optimal) path (shown by the solid line) following b until $t = 2$, after which the best plan is to wait for a until $t = 5$, after which the best plan is just to follow a.

Maximal ATF Learning for RTSIPP

RTAS can be seen as a minimal-learning approach to solving RTSIPP: it updates only a single state, but in return requires only a single pass over the open list. The next algorithm, Maximal ATF learning for RTSIPP (MaxATFS), is a maximal-learning approach, updating all states in the LSS in the style of LSS-LRTA* and PLRTA*. Like PLRTA*, we assume that the heuristic time-to-goal function returns separate admissible scalar values for h_s and f. MaxATFS makes two passes through the closed nodes in the LSS: in the first, the scalar h_s is updated, and in the second, the compound ATF f is updated. We consider paths through closed nodes as being composed of three segments. For example, consider the path b in Figure 2b. When updating closed state s_l, the three segments are: 1) (s_o, s_l), the path to the closed state (with time-dependent duration g); 2) (s_l, s_{f0}), the path from the closed state to the search frontier, which we will traverse in reverse through a Dijkstra-like back-up process; and 3) (s_{f_0}, s_g), the path from the search frontier to a goal, which is represented by the heuristic at the frontier.

Learning the Static Heuristic To learn the static heuristic, we restate the learning method of LSS-LRTA* for the SIPP state space, which we call LearnStatic. Algorithm 3 describes the algorithm, which differs from the original LSS-LRTA* learning stage only in terms of notation and being specific to the static component of the heuristic. The closed states are initialized to $h_s = \infty$ (line 3). Open is reordered by minimum h_s (line 4). We then do a Dijkstra-like traversal of the LSS, popping a minimum h_s frontier node (line 6) which contracts the search frontier; also we remove it from closed (line 8) indicating that it will not need further updating in this stage. We then visit each of the node's predecessors, updating their h_s (line 11) and (re)placing them on open (line 12) if we have found a lower value. Note that the size of closed is monotonically non-increasing, and all the h_s are monotonically decreasing. In the example shown in Figure 2b, we would learn $h_s(s_l)$ when we pop s_{f0} as the Dijkstra node, calculating $h_s(s_l) = 1 + 2$.

Algorithm 4: LearnDynamic

```
1: function LEARNDYNAMIC(open, closed)
2:     for s in closed do
3:         $f[s] \leftarrow \{\}$
4:     for n in open do
5:         $f[n] \leftarrow \{I[h_s(n)]\}$
6:     order open by increasing $\min_t f[s](t)$
7:     while open not empty and closed not empty do
8:         n ← pop(open)
9:         if state(n) in closed then
10:            remove state(n) from closed
11:        for all edge $e$ from parent $p'$ to state(n) do
12:            for all path $p$ to frontier in $f[s]$ do
13:                $A[p'] = A[p] \circ A[e]$
14:                if not $f[p']$ dominates $A[p']$ then
15:                    $f[p'] = f[p'] \cup \{A[p']\}$
16:                    add $p'$ to open maintaining ordering
```

Learning the Dynamic Heuristic To learn the dynamic heuristic, we again adapt LSS-LRTA*, except this time the Dijkstra style traversal backs up compound ATFs f (Equation 14) rather than scalar h_s. The LearnDynamic routine (Algorithm 4) initializes closed states to an empty compound ATF (line 3), and open states to the identity ATF, shifted by their static heuristic (line 5). Open is then sorted on the earliest (expected) arrival time at the goal, from the frontier (line 6). LearnDynamic then performs a Dijkstra-style traversal of the LSS where for each closed node, the ATFs in the compound ATF f's of all its children are composed with the edge from parent node to child node (line 13) and added to the parents compound ATF f (line 15). If the added ATF was not dominated (line 14), we (re)place p' in open (line 16).

Algorithm 5 shows MaxATFS, our maximal ATF learning algorithm for RTSIPP. MaxATFS updates the entire LSS: a partitioned static scalar h_s with LearnStatic, and dynamic

Algorithm 5: Maximal ATF learning for RTSIPP

```
1: function MAXATFS(cur, goal, b)
2:     if cur at goal then
3:         return success
4:     open, closed = NLASIPP(cur, goal, b)
5:     LEARNSTATIC(open, closed)
6:     LEARNDYNAMIC(open, closed)
7:     return BESTTLA(open)
```

Learning	PLRTS	RTAS	MedATFS	MaxATFS
Static	LSS	Single	LSS	LSS
Dynamic	0	Single	Single	LSS

Table 1: Algorithm Learning Characteristics

compound ATF f with LearnDynamic.

We also define a hybrid approach, Medium ATF learning for RTSIPP(MedATFS) which differs from MaxATFS in that does the dynamic learning from RTAS rather than LearnDynamic, causing it to update the static heuristic for the entire local search space, but the dynamic heuristic for only the current state. Table 1 summarizes the learning characteristics of the four algorithms.

MaxATFS backups maintain the consistency and admissibility of the base heuristic used at the search frontier.

Theorem 2. *If h_s is admissible/consistent and δ is monotonic then $h = f - g$ computed from the f learned by LearnDynamic is admissible/consistent.*

Proof. By induction. In the base case, frontier nodes have admissible/consistent h_s. The inductive step is line 13, we assume $A[p](t)$ is admissible/consistent, $A[e](t)$ is exact, so once we have learned from all the children of $a[p'](t)$, it must also be admissible/consistent. The process terminates because closed is monotonically non-decreasing in size and execution stops when it reaches size 0. □

Empirical Evaluation

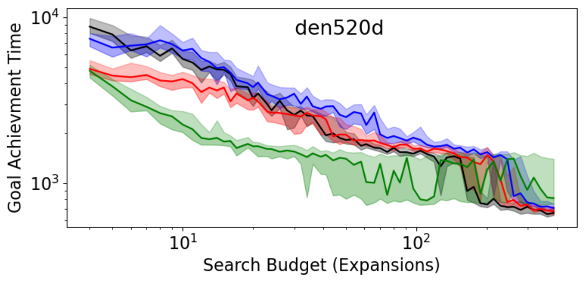
To evaluate the performance of these techniques, we ran a series of experiments on RTSIPP problems. We quantify the performance of the algorithms; PLRTS, RTAS, MedATFS, and MaxATFS as the expansion budget is varied. Our primary performance metric is the goal achievement time, which is the total time of the actions taken by the agent on its path to the goal including moving and waiting, excluding time spent solely on computation.

Experimental Set-Up

The agent has 4-way motion, and each movement takes 1 time unit. A static map is populated with random dynamic obstacles. At each unblocked grid cell, safe intervals are generated that have a length uniformly distributed from 'min duration' to 'max duration', and separated such that each location is unsafe 'ratio' percent of the time. Safe intervals are generated for each state until at least 10,000 time units, after which the state becomes unsafe.

Search time budgets in NLASIPP limit the number of expansions per move, which can be seen as corresponding to the amount of computation that can be accomplished per time unit. The algorithms are implemented in a shared, efficient C++ code base.[1] The NLASIPP implementation has an expansion rate of about 700,000 expansions/s and is shared by all four tested algorithms. Experiments were run one at a time on one of a cluster of identical computers, each with an Intel i3-12100 CPU and 64 GB of RAM.

[1]https://github.com/dwthomas/real-time-sipp



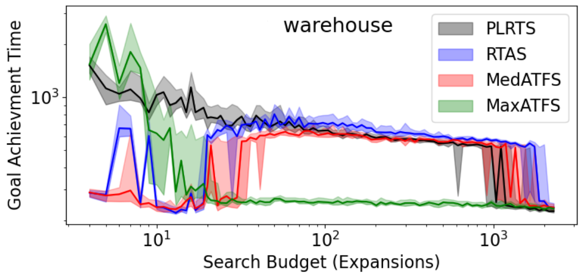


Figure 4: Goal achievement time results, median (solid line), 95% confidence interval(filled), limited to 10000 time units.

We test 4 static problem instances, using 32 random seeds, and logarithmically spaced expansion budgets from 4 to approximately the number required by SIPP to solve the instance. The static problem instances consist of a 2D grid with static obstacles and a start and goal location. Three of the static problem instances are scenarios from the Moving AI Lab Benchmarks: random512-25-0 (Figure 5a, cropped to show detail), den520d (Figure 5b), and warehouse-10-20-10-2-1 (Figure 5c) (Sturtevant 2012).

Random was selected to provide a map where the challenge is predominantly from the dynamic obstacles because there are only a few small local minima in the static map. Den520d is derived from a level from the game Dragon Age: Origins and is used to represent a situation where a game might need to navigate around dynamic obstacles. Warehouse was included to highlight the situation where the agent must navigate through thin corridors, that can become blocked. The cup map (Figure 5d) is a hand-crafted map with a single cup-shaped local minimum to challenge an algorithm to escape from large local minima. Random, den520d, and warehouse were generated with safe intervals from 500-1000 units long, cup 10-100. The den620d and cup maps used a ratio of 50%, random 25%, and warehouse 10%.

Results

Figure 4 shows some goal achievement time (in time units) results. On the den520d map (fig 4 top) we see similar performance between algorithms for very small (<5) budgets and very large budgets (>100). MedATFS performs significantly better than RTAS and the PLRTS for low budgets. This suggests that combining LSS-LRTA* style static learning and RTA* dynamic learning is beneficial beyond what

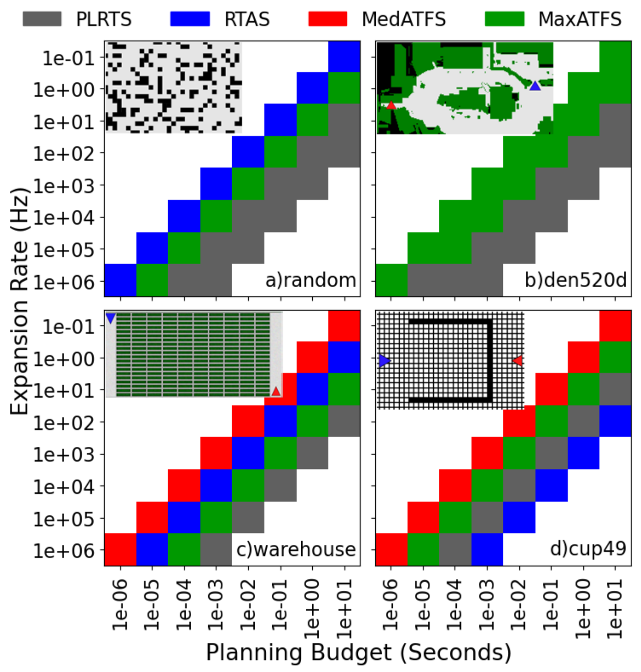


Figure 5: Best algorithm in simulated real-time experiments.

either provides alone. MaxATFS dramatically outperforms the other three methods. MaxATFS outperforming the static LSS-LRTA* methods (MedATFS, PLRTS) suggests that it is the dynamic portion of the learning that is critical.

Warehouse is an interesting setting, because there is an exploration/exploitation tradeoff when deciding which hallway to travel down. Whatever hallways you take, you would expect to encounter the same number of dynamic obstacles and have to wait the same amount of time on them. In the warehouse results (Figure 4 bottom) we see MaxATFS performing similarly to the PLRTS until a budget of around 10, where its performance dramatically improves. In contrast, RTAS and MedATFS perform much better than the PLRTS and MaxATFS until their performance dramatically worsens at budgets larger than 10. Until the budgets reach 1000, MaxATFS performs significantly better than the other three algorithms. The good performance of MaxATFS suggests that once the budget is greater than 10 MaxATFS is able to learn a dynamic heuristic that guides the search to select the best paths. In contrast, it appears that the less powerful dynamic heuristic learning of RTAS and MedATFS is learning a dynamic heuristic that guides the search to do more exploration, causing it to perform worse than with a tiny budget.

In experiments on the random map (not pictured), we observed similar behavior to den520d, except MaxATFS did not dramatically outperform MedATFS until a budget of 10. The transition at 10 may be the scale size of the random problem, where MaxATFS is able to perfectly guide the local search, while the pre-learning base heuristic is sufficient for the global search. In the experiments on the cups map (not pictured), MedATFS and MaxATFS do well learning a static heuristic in the presence of dynamic obstacles, and outperform the PLRTS with small expansion budgets. However here RTAS has a dramatic degradation in performance as the budget is increased, similar to the hump seen in the warehouse results. This suggests that it is having a similar issue with learning the heuristic, however, one that is addressed by a LSS-LRTA* style learning method for the static heuristic, where good performance in the warehouse map seemed to require LSS-LRTA* style learning for the dynamic heuristic as well. Overall, we find MaxATFS has a significantly better goal achievement time for most expansion budgets in the random, warehouse, and den520d maps.

It is not surprising that the algorithms that do more learning often perform better when given the same expansion budget. We now adjust the results to take into account the time overhead of each algorithm. Figure 5 plots which algorithm has the lowest goal achievement time for a given simulated real-time and simulated expansion rate. We measure the time spent planning, and learning during our experiments and extrapolate what the total per-iteration planning budget would have been, in seconds, if ASIPP had various expansion rates. Each cell is colored based on the algorithm with the lowest median goal achievement time over the 32 random seeds. We find that MedATFS or RTAS performs best when there is a limited planning budget, while MaxATFS performs best for moderate planning budgets. PLRTS performs the best when there is a large planning budget.

Related Work

Prior work in SIPP has included experiments run 'in real-time' (Narayanan, Phillips, and Likhachev 2012), including on robots (Phillips and Likhachev 2011). These run anytime or offline SIPP fast enough on certain problem instances to be used in real-time but are not true real-time heuristic search algorithms as the time they spend planning depends on the problem size. Sigurdson et al. (2018) describes a real-time heuristic search for multi-agent pathfinding, similar to SIPP but with multiple agents and discrete time steps.

Conclusion

We introduced the Real-time SIPP problem setting and showed how to adapt real-time heuristic search methods to efficiently handle dynamic environments using ideas from any-start-time planning. We defined three algorithms to learn consistent heuristics using ATFs. RTAS is an adaptation of RTA* to SIPP, which can learn without performing any backup operations. MaxATFS is a correct heuristic partitioning scheme for RTSIPP that learns a consistent heuristic for all states in the LSS. MedATFS is a hybrid approach that uses LSS-LRTA*-style learning for the static heuristic but the simpler RTA*-style method for the dynamic heuristic. We found that RTAS and MedATFS can outperform the static-only PLRTS algorithm, particularly with smaller search budgets, and MaxATFS outperforms the other three algorithms for a wide range of search budgets. Although making SIPP real-time turns out to be remarkably complex, the machinery we introduce is also likely to apply in other situations where the passage of time affects the value of acting, such as in situated planning.

Acknowledgments

We are grateful for support for this work from the United States National Science Foundation (NSF grant 2008594) and the United States-Israel Binational Science Foundation (BSF grant 2019730).

References

Cannon, J.; Rose, K.; and Ruml, W. 2014. Real-time Motion Planning with Dynamic Obstacles. *AI Communications*, 27: 345–362.

Kiesel, S.; Burns, E.; and Ruml, W. 2015. Achieving goals quickly using real-time search: experimental results in video games. *Journal of Artificial Intelligence Research*, 54: 123–158.

Koenig, S. 2001. Agent-Centered Search. *AI Magazine*, 22(4): 109–131.

Koenig, S.; and Sun, X. 2009. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, 18(3): 313–341.

Korf, R. E. 1990. Real-time Heuristic Search. *Artificial Intelligence*, 42: 189–211.

Narayanan, V.; Phillips, M.; and Likhachev, M. 2012. Anytime Safe Interval Path Planning for dynamic environments. In *Proceedings of IROS*, 4708–4715.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe interval path planning for dynamic environments. In *Proceedings of ICRA*, 5628–5635.

Sigurdson, D.; Bulitko, V.; Yeoh, W.; Hernández, C.; and Koenig, S. 2018. Multi-Agent Pathfinding with Real-Time Heuristic Search. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2): 144–148.

Thomas, D. W.; Shimony, S. E.; Ruml, W.; Karpas, E.; Shperberg, S. S.; and Coles, A. 2023. Any-Start-Time Planning for SIPP. In *Proceedings of the ICAPS-23 Workshop on Heuristics and Search for Domain-Independent Planning (HSDIP-23)*.